Struttura modulare ed a livelli di un sistema di calcolo

Modularità

Ha due dimensioni indipendenti

a) decomposizione verticale: per avere sistemi con funzionalità utente complesse non possiamo costruire tali funzionalità direttamente sulla macchina nuda ma dobbiamo utilizzare step (funzionalità) intermedi
b) decomposizione orizzontale: per quanto semplice sia ognuno dei vari passi intermedi dobbiamo, per efficienza di costruzione e di esecuzione, decomporlo in parti tra loro parzialmente indipendenti

Astrazione e concretizzazione

Ogni calcolatore, sistema di calcolo, sistema informatico può essere descritto come una **gerarchia di macchine virtuali** che

a) parte dalla macchina **fisica**

b) arriva fino a quella **applicativa** = che l'utente vede ed usa

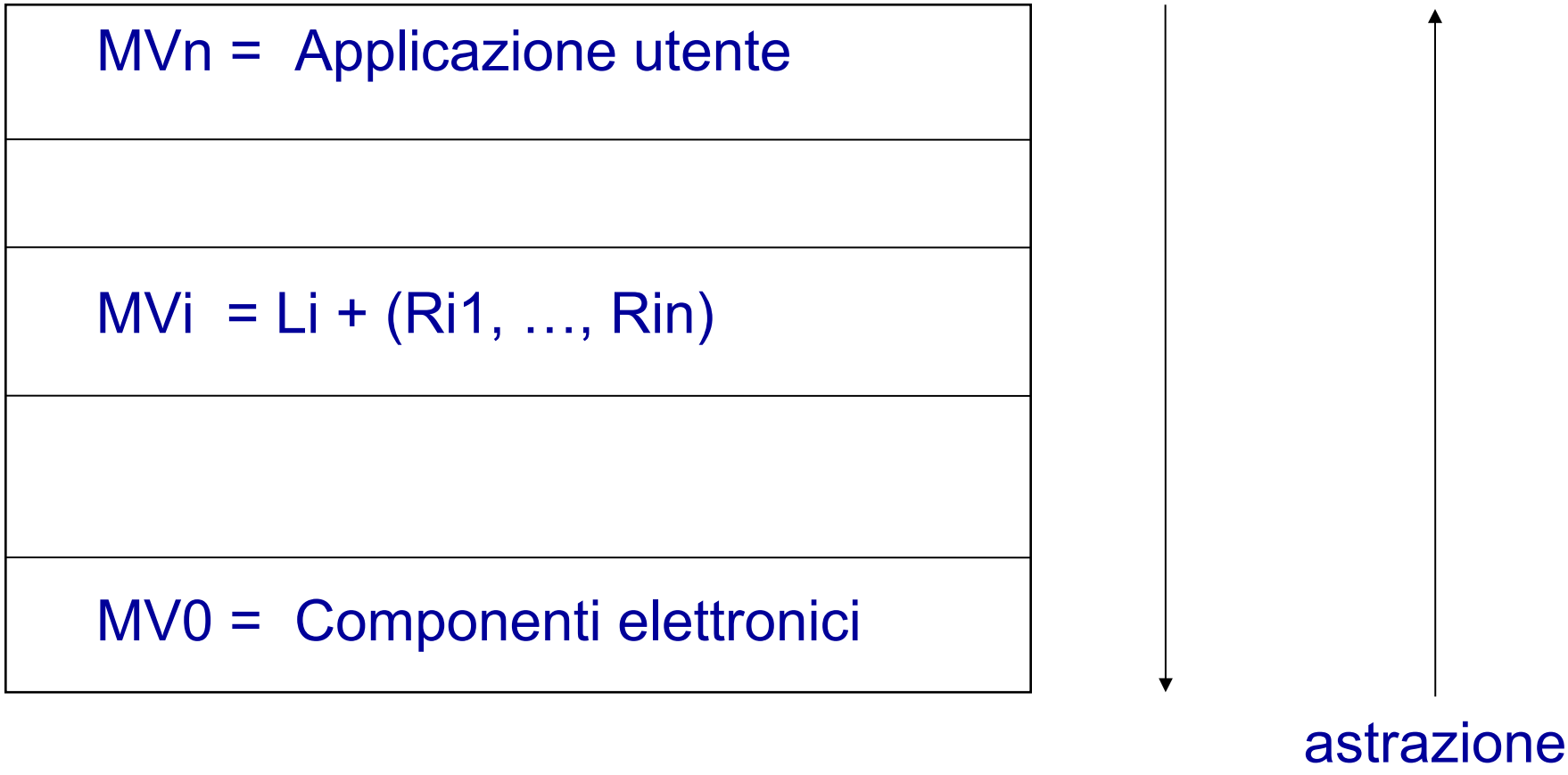
- Macchina virtuale = un linguaggio di programmazione ed un insieme di risorse per l'utente usa = un modo diverso di definire un linguaggio di programmazione
- Gerarchia di macchine virtuali = ogni macchina è definita a partire da quella sottostante e supporta quella sopra di essa

Il passaggio

- dalla macchina fisica a quella applicativa è detto di astrazione o di virtualizzazione
- dalla macchina applicativa a quella fisica è di concretizzazione

Gerarchia di macchine virtuali

concretizzazione



astrazione

Macchina virtuale

- Li = un linguaggio di programmmazione = un insieme di istruzioni
- Ri1, …, Rin = un insieme di risorse virtuali
- MVi = un insieme di programmi
  - scritti in Li
  - che operano su Ri1, …, Rin

Esempi

- Linguaggio del sistema operativo, file
- Linguaggio macchina, registri e posizioni di memoria
- SQL e database

Gerarchia di macchine -1

- La gerarchia permette di capire cosa fanno i programmi delle macchine virtuali
- In una gerarchia il programma scritto nel linguaggio Li della macchina MVi sulle risorse Ri1, …, Rin implementa il linguaggio di programmazione e le risorse della macchina virtuale MVi+1
- Abbiamo un approccio **modulare** per coprire il gap tra MV0 e MVn perché definiamo una gerarchia invece che una sola macchina
- Approccio top-down (concretizzazione) si parte da MVn, poi si definisce MVn-1 fino a MV0
- Approccio bottom-up (astrazione) si parte da MV0, poi si definisce MV1 fino a MVn
- Approccio middle-out si definisce MVi, poi da MVi+1 fino a MVn e da MVi fino a MV0

Approcci alla gerarchia

- Ci siamo ricondotti ricorsivamente al problema di mappare una macchina virtuale su un'altra =

Dati

- L_i e $R_{i1}, \ldots, R_{in}$ = insieme di operazioni e dati
- L_{i-1} e $R_{i-11}, \ldots, R_{i-1m}$ = insieme di operazioni e dati

Costruire 1) assumendo che 2) esista

Problema informatico comune, le due soluzioni tipiche sono

a) Compilazione

b) Interpretazione

hanno prestazioni diverse su tempo di esecuzione, costo, …

Approcci alla gerarchia - compilazione

L_i e R_i =(R_{i1}, …, R_{in}) macchina sorgente

L_{i-1} e R_{i-1}= (R_{i-11}, …, R_{i-1m}) macchina target della compilazione

Un programma sorgente scritto in L_i e che lavora su R_i viene tradotto in un programma target scritto in L_{i-1} e che lavora su R_{i-1} prima di essere eseguito = tradotto ***staticamente***

Caratteristiche

- A tempo di esecuzione la mcchina sorgente è sparita
- Il tempo (=costo) della traduzione viene pagato una sola volta, dipende unicamente dal numero di istruzioni del programma della mcchina sorgente
- Il tempo della traduzione non influenza il tempo di esecuzione
- il programma che effettua la traduzione viene detto ***compilatore*** del linguaggio sorgente nel linguaggio target
- indipendente da R_i = funziona qualunque sia questo insieme

Approcci alla gerarchia - interpretazione

L_i e R_i =(R_{i1}, …, R_{in}) macchina sorgente

L_{i-1} e R_{i-1}= (R_{i-11}, …, $R_{i-1m)}$ macchina target della interpretazione

Si scrive un programma nel linguaggio target (scritto in L_{i-1} e che lavora su R_{i-1}) che legge una alla volta le istruzioni del programma sorgente e la esegue. Traduzione ***dinamica*** nel linguaggio target = l'istruzione viene tradotta ogni volta che si esegue

Caratteristiche

- Il programma nel linguaggio target è un interprete del programma sorgente scritto nel linguaggio target
- A tempo di esecuzione il linguaggio sorgente è diventato un dato dell'interprete = distinzione tra programmi e dati è arbitraria
- Il tempo (=costo) della traduzione viene pagato tutte le volte che si esegue una istruzione viene eseguita = Il tempo della traduzione influenza il tempo di esecuzione

Differenza importante

- Per passare da Li a Li-1 l'interprete di Li è scritto in Li-1
- Per compilare Li in Li-1, uso un programma, **il compilatore, che può essere scritto in un qualunque linguaggio** perché non è necessario che la traduzione avvenga sul sistema dove il programma sarà eseguito (cross compilazione)
- E' possibile che il compilatore da Li a Li-1 sia scritto in Li, basta che io possa utilizzare una macchina **già costruita** che fornisce Li
- Mentre l'interpretazione deve avvenire sul sistema che mi interessa, la compilazione può avvenire su un qualunque sistema, poi si prende il risultato della compilazione, l'eseguibile e lo si porta sul sistema che si vuole utilizzare.

Terzo caso (quello reale)

L_i e $\mathbf{R}_i = (R_{i1}, \ldots, R_{in})$ macchina sorgente

L_{i-1} e $\mathbf{R}_{i-1} = (R_{i-11}, \ldots, R_{i-1m})$ macchina target

- Si introducono un linguaggio $L_{i, i-1}$ e delle risorse $\mathbf{R}_{i,i-1}$ $=(R^1_{i, i-1}, \ldots, R^w_{i, i-1})$ intermedie
- Intermedio dal punto di vista della astrazione tra L_i e L_{i-1}
- Spesso il linguaggio intermedio è una semplice estensione di L_{i-1}
- L'implementazione avviene in due passi
  - $L_i + (R_{i1}, \ldots, R_{in})$ compilazione $L_{i, i-1} + (R^1_{i, i-1}, \ldots, R^w_{i, i-1})$
  - Un interprete per $L_{i, i-1} + (R^1_{i, i-1}, \ldots, R^w_{i, i-1})$ scritto in $L_{i-1} + (R_{i-11}, \ldots, R_{i-1m})$
- Si compila una sola volta, le istruzioni dell'eseguibile sono tradotte (interpretate) ad ogni esecuzione dell'eseguibile

Terzo caso

- E' in pratica l'unica soluzione che si adotta per un qualunque linguaggio di programmazione
- Si parla di linguaggio interpretato se il linguaggio intermedio è molto vicino ad Li, compilato se è molto vicino ad Li-1
  - Java è interpretato, bytecode simile a programma sorgente, interprete JVM
  - C è compilato, interprete si occupa unicamente della gestione della memoria e di ingresso/uscita
- Tendenza è di privilegiare l'interpretazione perché cosi
  - il comportamento del programma può essere variato senza modificare il programma stesso
  - Non tutto l'interprete è presente in memoria, parte può essere caricata dinamicamente (DLL)
- In precedenza si tendeva a privilegiare compilazione per motivi di prestazioni (esecuzione più veloce)

Terzo caso

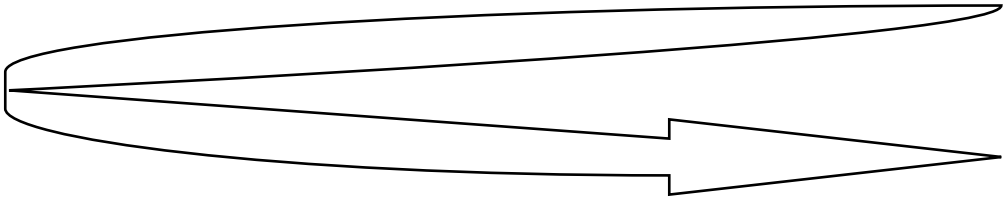
- L'interprete del linguaggio intermedio = supporto a tempo di esecuzione di L_i
- Quindi per implementare L_i si utilizza
  - Compilatore da L_i ad intermedio
  - Interprete del linguaggio intermedio = supporto a tempo di esecuzione di Li
- A tempo di esecuzione la macchina di L_i sparisce e resta solo l'interprete = a tempo di esecuzione resta solo supporto a tempo di esecuzione
- Se teniamo conto del fatto che abbiamo una gerarchia di livelli, a tempo di esecuzione, questa gerarchia diventa una gerarchia di supporti a tempo di esecuzione

Fondamentale

- Distinguere sempre
  - Il linguaggio sorgente da quello target
  - Il programma dell'interprete da quello da eseguire (da interpretare)
- Poiché si usa sempre un qualche interprete, esistono sempre due programmi in gioco
  - Da interpretare : può avere una struttura qualunque, le sue istruzioni vengono eseguite una alla volta ed operano sulle strutture dati del programma
  - Interprete: qualunque sia i (e quindi L_i e L_{i-1}) ha sempre una struttura

    Repeat
    leggi istruzione = fetch
    case istruzione letta *of* = decodifica
    …. ogni ramo invoca una procedura = esecuzione
    endcase
    Until programma finito
    detto ciclo di fetch-decodifica-esecuzione
- Confondere i due programmi porta ad errori drammatici

Gerarchia di macchine virtuali a tempo d'esecuzione



MVn = Applicazione utente
MVi = Li + (Ri1, …, Rin)
MV0 = Componenti elettronici

STEn
STEi
STE0= Componenti elettronici

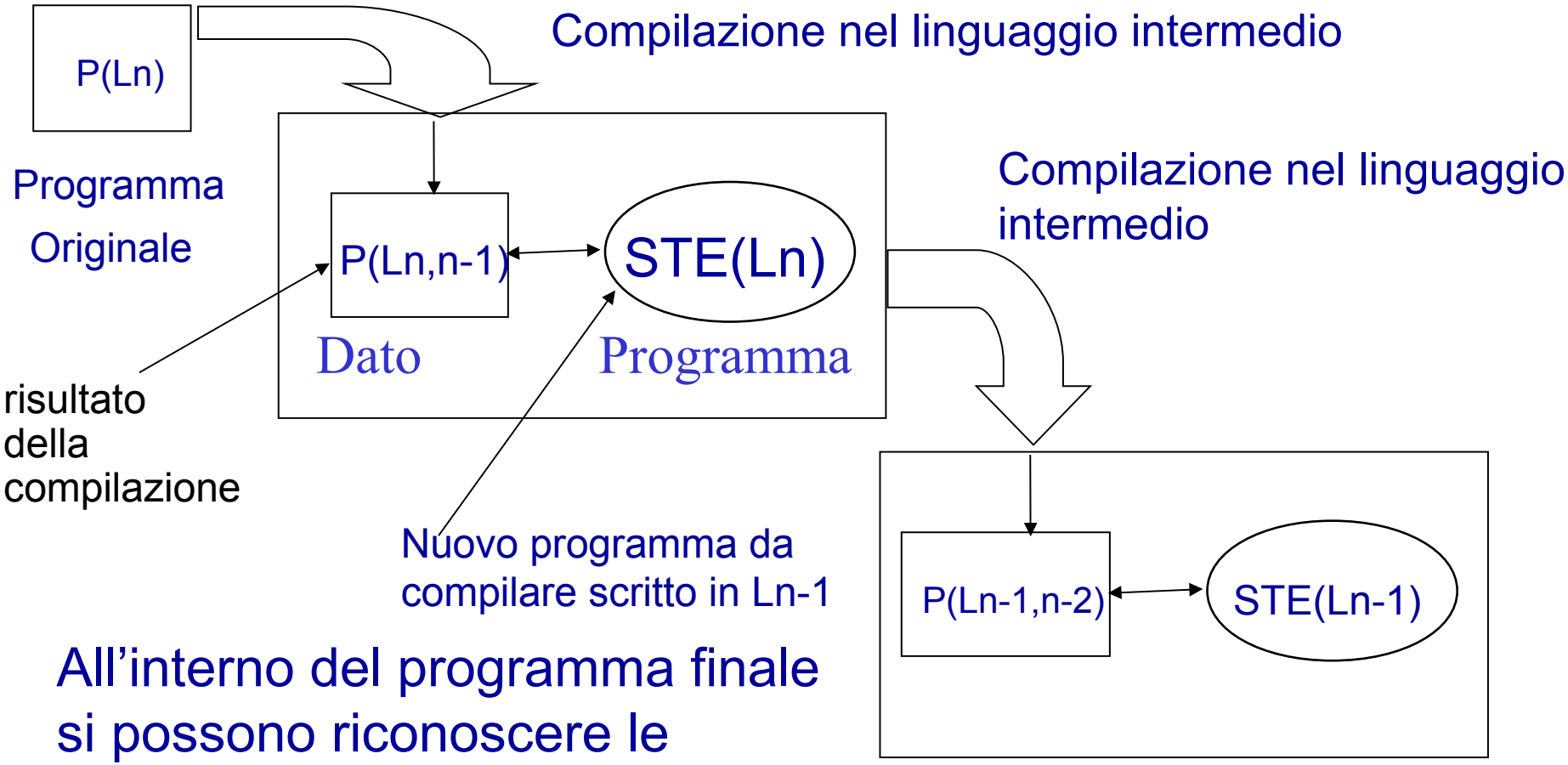
Prima dell'esecuzione

A tempo d'esecuzione

Ovviamente a tempo di esecuzione troveremo anche il programma prodotto dalle compilazioni e che i vari supporti devono interpretare

Il programma è un dato di questi interpreti

Gerarchia a tempo d'esecuzione



All'interno del programma finale si possono riconoscere le procedure dei vari interpreti

Il linguaggio macchina

- Abbiamo detto che c'è sempre un linguaggio intermedio, questo è vero per linguaggi utente o per il sistema operativo
- Non è vero per il linguaggio macchina
- Il linguaggio macchina è quel linguaggio
  - Sempre totalmente interpretato
  - Interprete è costruito dai componenti elettronici
- La definizione data è totalmente indipendente dai costrutti del linguaggio stesso = un qualunque linguaggio può essere il linguaggio macchina
- Teoricamente potrei avere un linguaggio macchina uguale al C o a Java poiché ciò che conta non sono le istruzioni e cosa fanno ma i due vincoli precedenti sulla realizzazione
- L'esperienza ha dimostrato che per avere sistemi con elevate prestazioni in termini di esecuzione dei programmi il linguaggio macchina deve essere semplice (RISC)
- Non conviene avere linguaggio macchina tipo C o Java perché sarebbe troppo complicato per una esecuzione veloce dei programmi

Termini e nozioni fondamentali

- Macchina virtuale
- Gerarchia delle macchine
- Compilazione
- Interpretazione
- Linguaggio intermedio
- Supporto a tempo di esecuzione
- Programma come dato
  - del compilatore
  - dell'interprete
- Ciclo di fetch-decodifica-esecuzione dell'interprete

Seconda dimensione della modularità

- Abbiamo visto come la gerarchia sia un modo di modularizzare l'implementazione (divide et impera) decomponendola in una sequenza di problemi più semplici ognuno dei quali richiede di realizzare un compilatore ed un interprete
- Noti i linguaggi intermedi, la costruzione dei compilatori e degli interpreti può avvenire in parallelo
- E' possibile riusare alcuni linguaggi con i relativi compilatori ed interpreti (caso tipico sistema operativo)
- Questa modularizzazione non è spesso sufficiente perché la realizzazione del singolo interprete rimane spesso estremamente complessa
- Occorre applicare un approccio modulare anche alla realizzazione del singolo interprete (decomposizione orizzontale dopo quella verticale dei livelli)

Moduli e cooperazione tra moduli - I

- Il programma del singolo interprete non può essere visto come un programma monolitico
- Per ridurne la complessità occorre decomporlo in moduli, ognuno dei quali esegue alcuni dei compiti dell'interprete
- I moduli si scambiano informazioni per implementare l'algoritmo complessivo dell'interprete
- Lo scambio di informazioni avviene mediante una struttura di interconnessione
- Interprete = Insieme di moduli + struttura di interconnessione
- Ogni modulo ha un proprio programma che ne descrive il comportamento
- Globalmente ogni modulo è quindi una entità
  - Autonoma = ha un proprio programma che ne descrive il comportamento ed esegue una istruzione solo se è prevista nel proprio programma
  - Sequenziale = esegue una istruzione alla volta
  - Cooperante = per eseguire il proprio programma scambia informazioni con gli altri moduli

Moduli e cooperazione tra moduli - II

Autonomia+Sequenzialità+Cooperazione

- non è possibile forzare un modulo a cooperare
- ad un certo istante o coopera o lavora localmente

L'autonomia richiede che i moduli siano, almeno concettualmente, in esecuzione contemporanea perché se un modulo fosse eseguito solo quando un altro lo invoca, se ne limiterebbe l'autonomia

Non basta un linguaggio sequenziale occorre un linguaggio concorrente o distribuito in cui si creano più moduli in esecuzione e che cooperano per implementare le funzioni di interesse = programma concorrente o distribuito in base al sistema che lo esegue

Nel caso di nostro interesse il supporto a tempo di esecuzione è concorrente e quindi scritto con un linguaggio di programmazione concorrente.

Moduli e parallelismo

- Ad ogni istante una istruzione per ogni modulo può essere in esecuzione = il numero di moduli si indica come grado di parallelismo massimo del sistema
- Il grado di parallelismo è un indicatore (è correlato a) il grado di efficienza del sistema complessivo
- Se si vuole costruire un interprete efficiente la strategia più rigorosa ed efficace è quella che aumenta il numero di moduli e quindi il parallelismo tra istruzioni in esecuzione

Esempio di cooperazione

Nel caso di supporto a tempo di esecuzione, il programma concorrente è un interprete = un case in un ciclo

- Un modulo C esegue il case
- In base al ramo trasmette l'istruzione ad un altro modulo Rj
- Rj esegue l'istruzione e trasmette il risultato a C

Vantaggi

- Posso aggiungere e togliere moduli se aggiungo o tolgo istruzioni al linguaggio da interpretare
- C non conosce le singole istruzioni e posso riusarlo per linguaggi diversi
- Posso riusare moduli se le stesse istruzioni compaiono in linguaggi diversi
- Inizializzazione e gestione di strutture dati può avvenire contemporaneamente in moduli diversi

Una conseguenza importante

Per descrivere un sistema al livello di interesse, occorre descrivere una macchina virtuale

- Descrivere la macchina virtuale equivale a descrivere i moduli che la compongono
- Descrivere i moduli che la compongono equivale a descrivere i programmi dei singoli moduli
- La descrizione mediante programma è una descrizione generale applicabile ad un qualunque sistema che poi può essere realizzato in modo molto diverso, ad esempio con componenti elettronici
  Se so programmare allora so descrivere moduli

Implementazione di modulo

- Il termine modulo è un termine generico a cui corrispondono nozioni diverse nelle diverse macchine virtuali

- Al livello hardware modulo = singolo componente
- Al livello firmware modulo = unità di elaborazione

  unità = processore, memoria principale, cache, unità di I/O in base al programma eseguito
- A livello di sistema operativo modulo = processo (se decomposto in più thread equivale a più moduli)
- A livello applicazione modulo = processo/thread

In ognuna delle specializzazioni del concetto di modulo possiamo trovare il programma che ne determina il comportamento, i dati su cui lavora, la cooperazione con altri moduli

Implementazione

- Nel seguito discutiamo le proprietà dei moduli generali, che valgono indipendentemente da come sono realizzate
- Analizziamo quindi i problemi comuni alle varie implementazioni e le varie caratteristiche di interesse dei moduli trascurando la loro implementazione
- Non è possibile descrivere l'implementazione poiché essa dipende dal livello considerato
- Specifiche implementazioni verranno descritte nel seguito del corso nel caso di
  - Moduli = Unità di Elaborazione
  - Moduli = Processi del Sistema Operativo

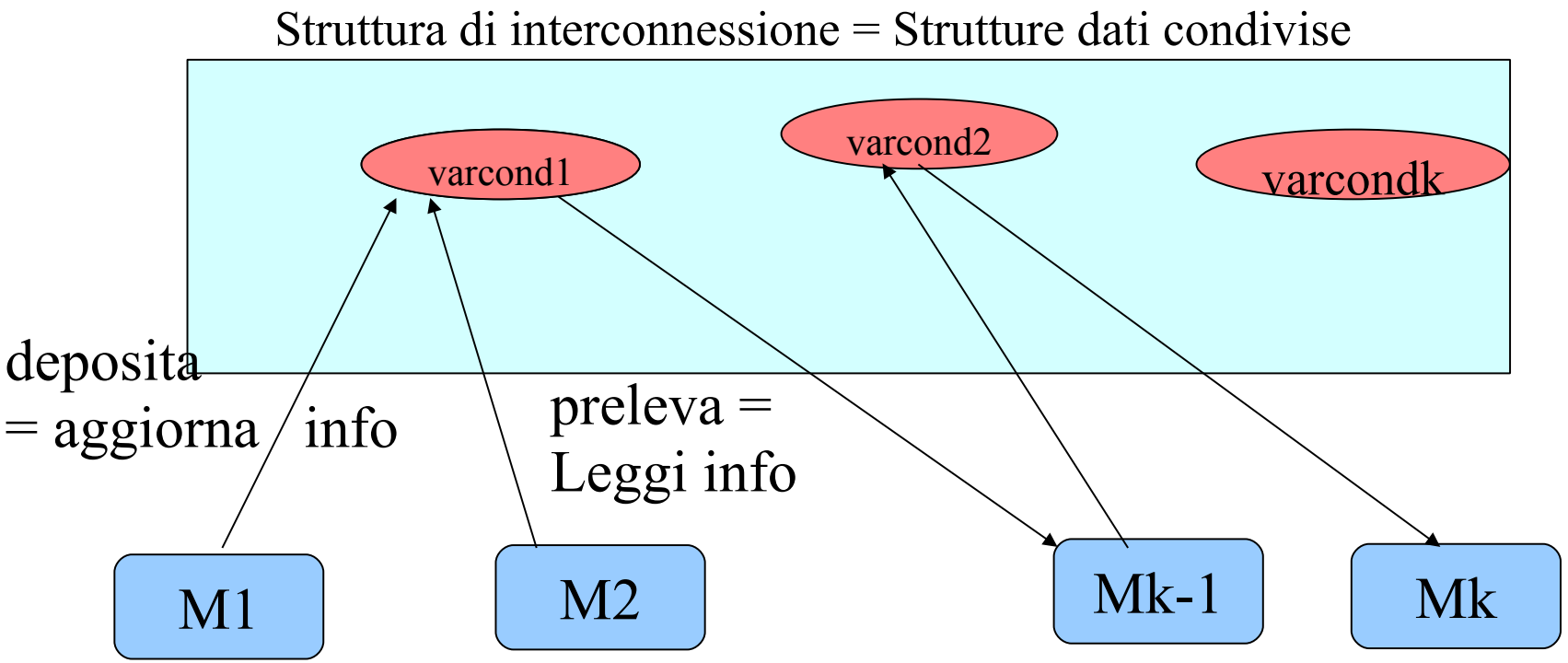
Cooperazione

- Il fatto che abbiamo decomposto un sistema complesso in più moduli fa si che tali moduli debbano lavorare insieme per costruire il sistema complessivo
- I moduli quindi sono correlati e non del tutto indipendenti
- Correlazione o cooperazione implica uno scambio di informazioni in cui un modulo
  - riceve delle informazioni da un altro
  - le elabora localmente
  - trasmette i risultati dell'elaborazione ad uno o più altri moduli
- La cooperazione ha delle ripercussioni sull'organizzazione dei singoli moduli e genera dei vincoli di sincronizzazione
- Sincronizzazione = una istruzione in un modulo può essere eseguita solo se alcune istruzioni di altri moduli sono state eseguite = un ordinamento globale sulle istruzioni dei moduli

Come implementare la cooperazione

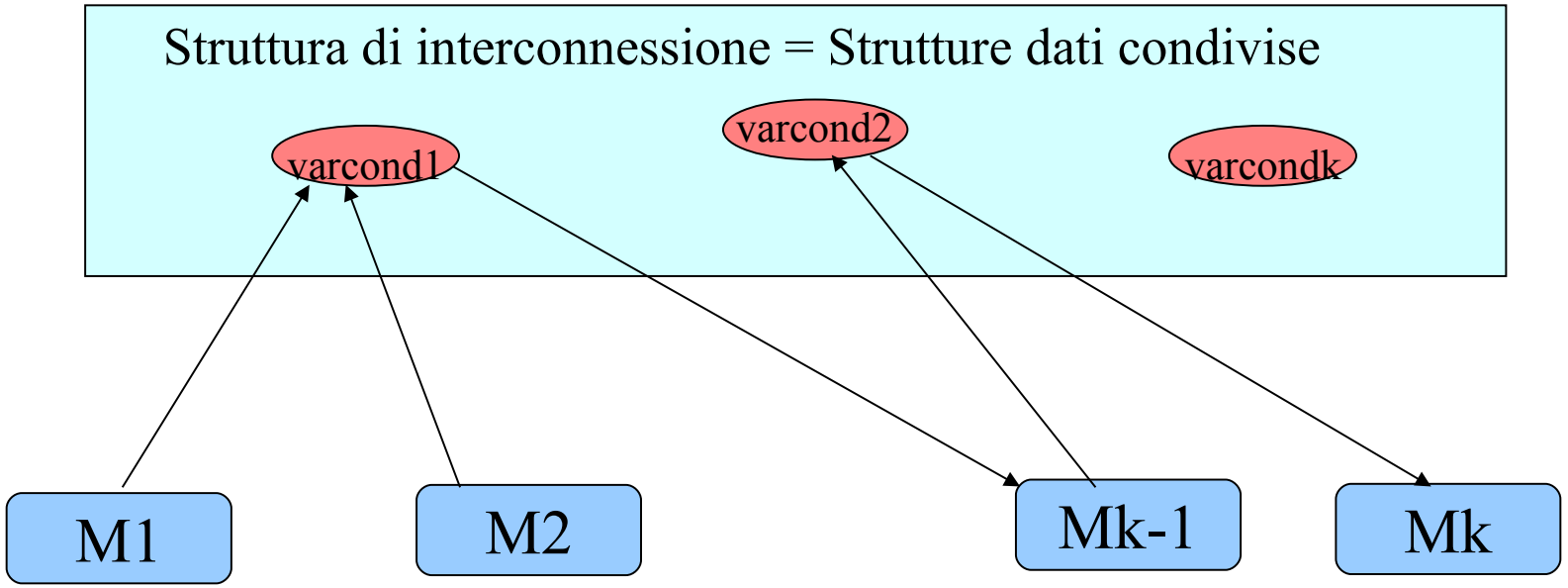
- La cooperazione può avvenire in 2 modi diversi che portano a definizioni diverse della struttura di interconnessione
- **Ambiente globale**: la struttura di interconnessione è formata da strutture dati condivise che vengono aggiornate e lette da più moduli. La cooperazione avviene quindi lavorando su strutture dati comuni a più moduli. Ovviamente ogni modulo ha le proprie strutture dati private (Metodi serializer di Java)
- **Ambiente locale**: i moduli cooperano unicamente scambiandosi delle informazioni, dei valori. Un modulo A trasmette ad un modulo B un valore M che B assegna ad una propria variabile locale = assegnamento distribuito in cui un modulo definisce un valore e l'altro la variabile che deve ricevere il valore. In questo caso non ci sono strutture dati condivise ma ogni struttura dati è privata ad un qualche modulo

Ambiente Globale



Alcuni moduli depositano le informazioni nelle strutture dati altri moduli le prelevano, occorre garantire la mutua esclusione nell'esecuzione delle operazioni

Ambiente Globale



Il problema fondamentale = atomicità o mutua esclusione:

i) un solo modulo alla volta può modificare la struttura condivisa

ii) ogni operazione su una struttura non può essere interrotta da altre operazioni sulla stessa struttura

Esempio Ambiente Globale

Modulo 1

```
...
x=f(...);
y=g(...);
....
varcond1=x;
varcond2=y;
w=h(x, y);
t=varcond3;
z=varcond4;
```

Modulo 2

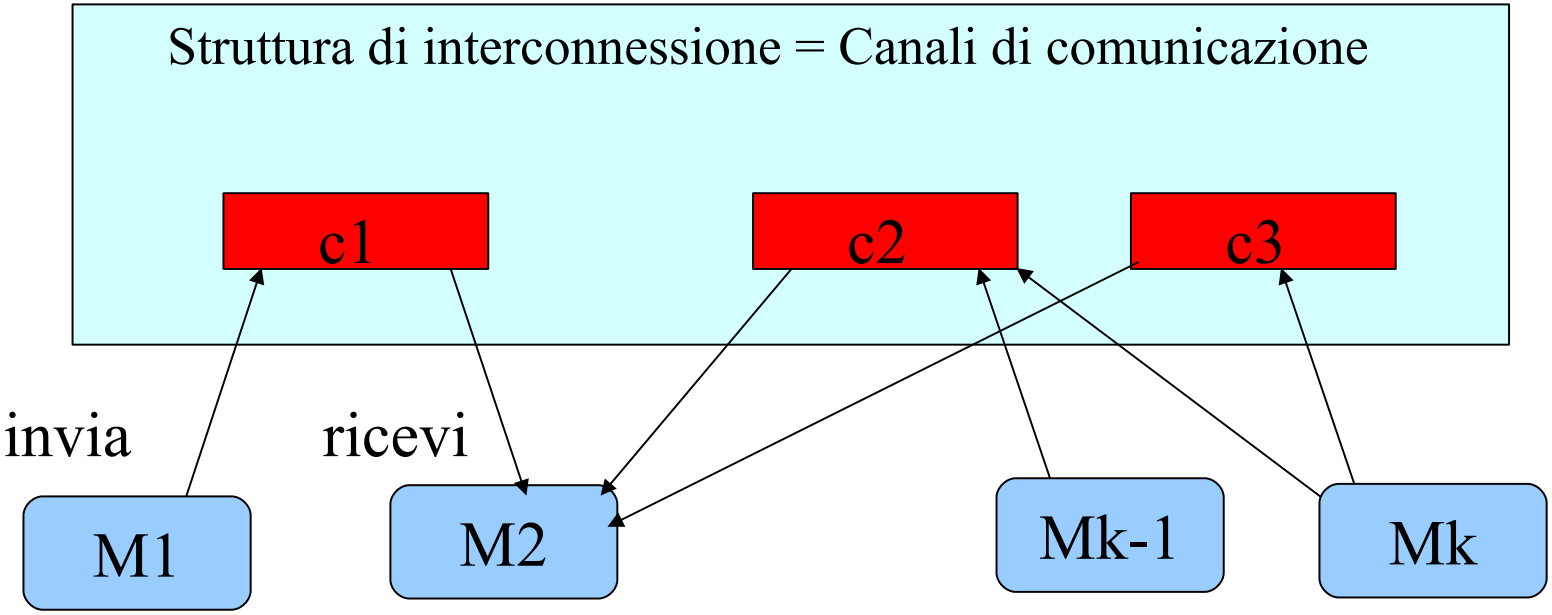
```
a=varcond1;
b=l(a);
varcond3=a;
```

Modulo 3

```
c=varcond2;
d=m(c);
varcond4=d;
```

a) grado di parallelismo 3= numero di moduli attivi contemporaneamente = 3 calcoli eseguiti contemporaneamente
b) operazioni su $varcond_i$ sono atomiche o in mutua esclusione

Ambiente locale



Un canale trasmette una informazione tra il modulo che la invia il mittente, e quello che la riceve, il destinatario

Tipi di canali - I

- Un canale è detto simmetrico se trasporta l'informazione tra un solo mittente ed un solo destinatario, comunicazione simmetrica
- Un canale asimmetrico trasporta l'informazione tra più di due processi
  - Asimmetria in ingresso: più moduli possono trasmettere le informazioni ma solo uno la riceve. Le informazioni rimangono distinte, chi riceve ogni volta riceve l'informazione trasmessa da un altro modulo
  - Asimmetria in uscita: più moduli possono ricevere le informazioni.
    - Non deterministico; uno solo tra quelli che la possono ricevere la riceve effettivamente (quello che la richiede per primo, uno scelto a caso ...)
    - A diffusione o broadcast: tutti ricevono una copia del valore trasmesso
- In generale c'è un solo tipo di asimmetria per canale perchè altrimenti non si controlla la cooperazione tra moduli

Esempio Ambiente Locale

Modulo 1

```
...
x=f(...);
y=g(...);
....
invia x a 2;
invia y a 3
w=h(x, y);
ricevi t da 2;
ricevi z da 3;
```

Modulo 2

```
ricevi a da 1
b=l(a);
invia b a 1;
```

Modulo 3

```
ricevi c da 1;
d=m(c);
invia d a 1;
```

canali implementano un assegnamento distribuito in cui uno dei moduli fornisce un valore e l'altro una variabile

Ambiente locale vs globale

- Per i livelli di sistema che interessano a questo corso,la soluzione ad ambiente globale non è sufficientemente generale
- Infatti se la macchina virtuale che vogliano progettare/analizzare/ modificare è quella in cui i *moduli sono realizzati come unità di elaborazione l'unica soluzione possibile è la cooperazione ad ambiente locale* poiché in questo caso
  - non è possibile separare la memoria dalle unità, la memoria può esistere una memoria solo all'interno di una unità
  - quindi ogni struttura dati è privata di una unità
- Nelle macchine virtuali di livelli superiori invece entrambe le soluzioni sono possibili
- Di conseguenza in questo corso considereremo il solo caso dell'ambiente locale
- Nel caso di sistemi operativi si considerano invece entrambi i tipi di cooperazione

Peculiarità del linguaggio macchina

- È interpretato
- L'interprete è implementato da moduli realizzati da componenti elettronici
- I moduli dell'interprete del linguaggio possono cooperare solo ad ambiente locale perchè sono realizzati a partire da componenti elettronici

Vincoli dovuti alla maggiore concretizzazione del linguaggio macchina

Sincronizzazione e grado di parallelismo

- La sincronizzazione introducendo dei vincoli sulla esecuzione delle istruzioni può ridurre
  - il grado di parallelismo e quindi
  - l'efficienza
- Il numero di moduli è quindi un estremo superiore sul grado di parallelismo
- Il grado di parallelismo effettivo richiede un'analisi della cooperazione e del tipo di sincronizzazione,
  - non è ovvio conoscere il grado di parallelismo
  - è ovvio conoscere il numero di moduli